Linear-Time Version of Holub's Algorithm for Morphic Imprimitivity Testing

Tomasz Kociumaka[1], Jakub Radoszewski[1],
Wojciech Rytter[1,2,⋆], and Tomasz Waleń[3,1,⋆⋆]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw,
Banacha 2, 02-097 Warsaw, Poland
{kociumaka,jrad,rytter,walen}@mimuw.edu.pl

[2] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University,
Chopina 12/18, 87-100 Toruń, Poland

[3] Laboratory of Bioinformatics and Protein Engineering,
International Institute of Molecular and Cell Biology in Warsaw, Poland,
Ks. Trojdena 4, 02-109 Warsaw, Poland

Abstract. Stepan Holub (Discr. Math., 2009) gave the first polynomial algorithm deciding whether a given word is a nontrivial fixed point of a morphism. His algorithm works in quadratic time for large alphabets. We improve the algorithm to work in linear time. Our improvement starts with a careful choice of a subset of rules used in Holub's algorithm that is necessary to grant correctness of the algorithm. Afterwards we show how to choose the order of applying the rules that allows to avoid unnecessary operations on sets. We obtain linear time using efficient data structures for implementation of the rules. Holub's algorithm maintains connected components of a graph corresponding to specially marked positions in a word. This graph is of quadratic size for large alphabet. In our algorithm only a linear number of edges of this conceptual graph is processed.

1 Introduction

We consider finite words that are fixed points of morphisms. A word $w \in \Sigma^*$ is called a fixed point of a morphism $h : \Sigma^* \to \Sigma^*$ if $h(w) = w$. We say that w is a *trivial* fixed point of h if $h(a) = a$ for every letter actually occurring in w. The problem of finding all fixed points of a morphism was already studied in [5, 6, 11]. A more difficult problem of finding for a given word $w \in \Sigma^*$ a morphism h such that w is a non-trivial fixed point of h was first studied by Holub [8, 7]. The problem turned out to have a polynomial-time solution although a similar problem of finding for a given pair of words w, v a morphism such that $h(w) = v$ was shown to be NP-complete [2]. A word w is called *morphically imprimitive* if there exists a morphism h such that w is a non-trivial fixed point of h. Otherwise, w is called *morphically primitive.*

⋆ The author is supported by grant no. N206 566740 of the National Science Centre

⋆⋆ The author is supported by Iuventus Plus grant (IP2011 058671) of the Polish Ministry of Science and Higher Education

Example 1. The word $w = aeecbdebaabdebeec$ is morphically imprimitive; it is a non-trivial fixed point of the following morphism h:

$$h:\ a \to a, \quad b \to \varepsilon, \quad c \to eec, \quad d \to bdeb, \quad e \to \varepsilon.$$

On the other hand, the word $w' = aeecbdebaabdebec$ is morphically primitive.

Let w be a word of length n over an alphabet Σ of size m. We assume that Σ is an *integer* alphabet, i.e. each letter $a \in \Sigma$ has a unique integer identifier of magnitude $n^{O(1)}$. The **main problem** considered here is to decide, for the given word w, whether it is morphically imprimitive and, if so, find a morphism h such that w is a non-trivial fixed point of h.

Holub [8] presented an $O((m+\log n)\cdot n)$ time algorithm for this problem and more recently Matocha and Holub [10] slightly improved the time complexity to $O(m \cdot n)$. Both solutions are potentially quadratic if $m = \Theta(n)$. We give an algorithm which works in $O(n)$ time for arbitrary integer alphabet.

The main components of Holub's algorithm are two sets $L, R \subseteq \{0, \ldots, n\}$. The algorithm performs implicitly quadratically many insertions into those sets in worst case. Each insertion can be either useless, when the inserted element is already in the set, or useful, otherwise. The crucial improvements to the Holub's algorithm presented in this paper are:

1. Reduction of the number of insertions into L to $O(n)$, this is obtained by changing the logics of basic operations performed in Holub's algorithm.
2. Reduction of the number of useless insertions into R to $O(n)$ using efficient data structures.

2 Preliminaries

By $Occ(a, w)$ (or simply $Occ(a)$) we denote the set of positions in w where the letter a occurs. We also denote $|w|_a = |Occ(a, w)|$. The set of letters actually occurring in w is denoted by $alph(w)$.

Let $F = (w_1, \ldots, w_k)$ be a factorization of the word w, $w = w_1 \ldots w_k$. We say that $e \in alph(w)$ is a *key-letter* if for each $i, j \in \{1, \ldots, k\}$ we have

$$|w_i|_e \le 1 \quad \text{and} \quad (|w_i|_e = |w_j|_e = 1 \text{ implies that } w_i = w_j).$$

In other words, each factor contains exactly one occurrence of a key-letter and factors are determined by the key-letters.

If e is a key-letter, let $F_e = w_i$, where $|w_i|_e = 1$. We say that e is a *standard* key-letter if no key-letter occurs in F_e before the occurrence of e.

We say that F is a *morphic* factorization if each factor w_i contains a key-letter. Let $\mathcal{E}$ be the set of standard key-letters of a morphic factorization F. Then setting $h(e) = F_e$ for $e \in \mathcal{E}$ and $h(a) = \varepsilon$ for $a \notin \mathcal{E}$ yields a morphism h such that $h(w) = w$. A morphism obtained this way is called a *standard* morphism of w. The elements of $\mathcal{E}$ are called *expanding* letters and the elements of $alph(w) \setminus \mathcal{E}$ are called *mortal* letters. Holub [8] proved the following equivalent condition on when a word is morphically imprimitive:

Lemma 2 (Theorem 1 in [8]). *A word w is morphically imprimitive if and only if it has a morphic factorization F which is nontrivial, that is, has less than $|w|$ factors.*

Let $w[i]$ denote the i-th letter of w (for $1 \le i \le n$) and $w[i,j]$ denote the word $w[i]w[i+1]\dots w[j]$. Let $\{0,1,\dots,n\}$ be the set of *inter-positions* of w located in the beginning of the word ($i = 0$), between any two letters of the word ($1 \le i < n$) or at the end of the word ($i = n$).

For a letter $a \in alph(w)$, we define the *right range* of a as a word $\mathbf{r}_a$ such that $a\mathbf{r}_a$ is the longest common prefix of all suffixes of w starting with a, and similarly the *left range* $\mathbf{l}_a$ such that $\mathbf{l}_a a$ is the longest common suffix of all prefixes of w ending with a. We denote $\mathbf{n}_a = \mathbf{l}_a a \mathbf{r}_a$. Let $r_a = |\mathbf{r}_a|$, $l_a = |\mathbf{l}_a|$ and $n_a = |\mathbf{n}_a|$.

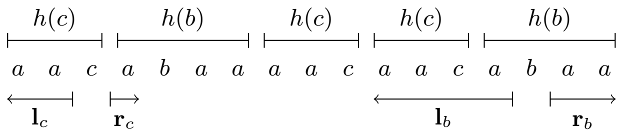

Fig. 1. For this word $\mathbf{n}_a = a$, $\mathbf{n}_b = aacabaa$ and $\mathbf{n}_c = aaca$. This word is morphically imprimitive due to the morphism $h(a) = \varepsilon$, $h(b) = abaa$, $h(c) = aac$

Denote by $\prec$ an ordering of $alph(w)$ according to $|w|_a$: $a \prec b$ if and only if $|w|_a < |w|_b$. For any $0 \le i < j \le n$, define $\alpha(i,j) = \min(alph(w[i+1,j]))$, where the minimum is taken according to the order $\prec$. If there are multiple equal letters, we choose the one with the leftmost first occurrence within $w[i,j]$.

Holub's algorithm for testing morphic imprimitivity can be stated as follows. The paper [8] provides a simple algorithm that recovers the morphism from the sets E, L, R.

Algorithm VERSION1: Holub's algorithm.

```
Maintain a triple of sets $(E, L, R)$, initially equal to $(\emptyset, \emptyset, \emptyset)$, and use the
following set of rules (a)-(e) to extend these sets until $E = alph(w)$ or
none of the actions alters the triple:

(a) $L := L \cup \{0, n\}$; $R := R \cup \{0, n\}$
(b) **if** $w[i] \in E$ **then** $L := L \cup \{i-1\}$; $R := R \cup \{i\}$
(c) **if** $w[i,j] = \mathbf{n}_a$ *for some* $a \in E$ **then** $R := R \cup \{i-1\}$; $L := L \cup \{j\}$
(d) **if** $w[i,j] = w[i',j'] = \mathbf{n}_a$ *for some* $a \in E$ **then**
      – **if** $i + k \in R$ *for some* $-1 \le k \le j - i$ **then** $R := R \cup \{i' + k\}$
      – **if** $i + k \in L$ *for some* $-1 \le k \le j - i$ **then** $L := L \cup \{i' + k\}$
(e) **if** $i < j$ **and** $i \in L$ **and** $j \in R$ **then** $E := E \cup \{\alpha(i,j)\}$

**if** $E \ne alph(w)$ **then return** *true* **else return** *false*
```

Let h be a standard morphism of w. Let $\mathcal{E}_h$ denote the set of expanding letters of h. Moreover, let us define two subsets of inter-positions, $\mathcal{L}_h$ of *left* inter-positions and $\mathcal{R}_h$ of *right* inter-positions:

$$\mathcal{L}_h = \{i : |h(w[1,i])| \le i\}, \qquad \mathcal{R}_h = \{i : |h(w[1,i])| \ge i\}.$$

Definition 3. *A triple* (E, L, R) *is called* correct *if* $E \subseteq \mathcal{E}_h$, $L \subseteq \mathcal{L}_h$ *and* $R \subseteq \mathcal{R}_h$ *for any standard morphism* h.

Lemmas 4, 6, 7 and 8 of [8] can be stated succinctly as the following fact:

Fact 4 *Extending a correct triple using any of the rules (a)-(e) leads to a correct triple. In particular, if any sequence of actions corresponding to (a)-(e) leads to* $E = alph(w)$ *then* w *is morphically primitive.*

3 The structure of our algorithm

Now we describe the set of rules used in our algorithm, which is a subset of rules from Holub's algorithm, and show that this set suffices to construct an algorithm for testing morphic primitivity (see the following Lemma 6 and Theorem 7).

Let us introduce some notation. Let A be a set of integers. We define the predecessor in A and successor in A in a standard way:

$$succ_A(x) = \min\{y : y \in A, y > x\}, \quad pred_A(x) = \max\{y : y \in A, y < x\}.$$

We assume $\min \emptyset = \infty$, $\max \emptyset = -\infty$.

Let $Occ(E) = \{i : w[i] \in E\}$ be the set of occurrences of expanding letters in w, we call them *expanding occurrences*. We slightly abuse the notation and write $succ_E$ and $pred_E$ instead of $succ_{Occ(E)}$ and $pred_{Occ(E)}$.

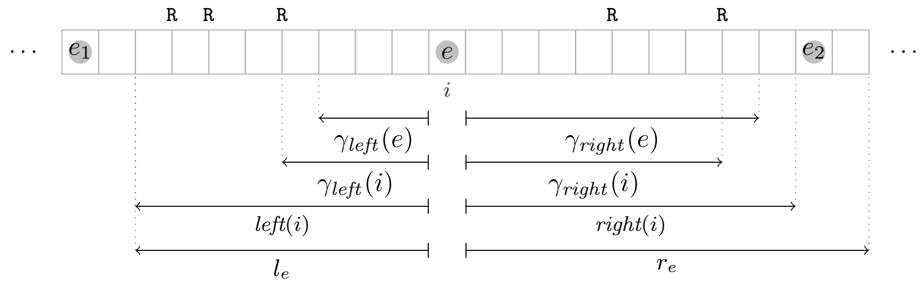

Fig. 2. Illustration of the main notions of the algorithm

The bottleneck of Holub's algorithm is performing the rule (d). Our crucial improvement, which makes linear time implementation possible, is twofold. First, we shrink the ranges so that they never exceed other expanding positions. This way, each inter-position lies in at most two ranges — originating at the closest expanding positions — and consequently propagation is simpler to control. We introduce *left* and *right* functions, defined for $i \in Occ(E)$:

$$left(i) = \min(l_{w[i]},\, i - pred_E(i) - 1), \quad right(i) = \min(r_{w[i]},\, succ_E(i) - i - 1).$$

Secondly, we only propagate elements of R from selected inter-positions. For $i \in Occ(E)$, define:

$$\gamma_{left}(i) = i - pred_R(i) - 1, \quad \gamma_{right}(i) = pred_R(i + right(i) + 1) - i.$$

Less formally, $\gamma_{left}(i)$ shows the location of the rightmost element of R before the position i, while $\gamma_{right}(i)$ points to the rightmost R within the range of the position i (see Fig. 2). We extend these definitions to expanding letters:

$$\gamma_{left}(e) = \min\{\gamma_{left}(i) : i \in Occ(e)\}, \quad \gamma_{right}(e) = \max\{\gamma_{right}(i) : i \in Occ(e)\}.$$

Thus we arrive at the following Algorithm Version2.

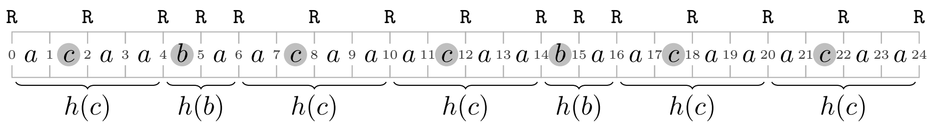
R R R R R R R R R R R R R R R
0 a 1 c 2 a 3 a 4 b 5 a 6 a 7 c 8 a 9 a 10 a 11 c 12 a 13 a 14 b 15 a 16 a 17 c 18 a 19 a 20 a 21 c 22 a 23 a 24
h(c) h(b) h(c) h(c) h(b) h(c) h(c)

Fig. 3. In the above word $\mathbf{n}_a = a$, $\mathbf{n}_b = acaabaacaaacaa$ and $\mathbf{n}_c = acaa$. Let $E = \{b, c\}$, these letters are marked in the figure. We have $\gamma_{left}(b) = 0$, $\gamma_{right}(b) = 1$, $\gamma_{left}(c) = 1$, $\gamma_{right}(c) = 2$. The morphic factorization is obtained by cutting the word at inter-positions $\{i - 1 - \gamma_{left}(w[i]) : i \in Occ(E)\}$; thus $h(b) = ba$, $h(c) = acaa$, $h(a) = \varepsilon$

Algorithm Version2

Perform, in any order, the following operations on a triple of sets (E, L, R), initially equal to $(\emptyset, \emptyset, \emptyset)$, until none of the operations alters the triple (note that the operations (b'), (c'), (d'), (e') together perform only a subset of actions from operations (b), (c), (d), (e) in Holub's algorithm):

```
(a)  $L := L \cup \{0, n\}$; $R := R \cup \{0, n\}$
(b') **if** $w[i] \in E$ **then** $R := R \cup \{i\}$
(c') **if** $w[i] \in E$ **then** $R := R \cup \{i - left(i) - 1\}$; $L := L \cup \{i + right(i)\}$
(d') **if** $w[i] \in E$ **then** $R := R \cup \{i - 1 - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])\}$
(e') **if** $i < j$ **and** $succ_R(i) = j$ **and** $pred_L(j) = i$ **and**
     $\{w[i+1], \ldots, w[j]\} \cap E = \emptyset$ **then**
         $E := E \cup \{\alpha(i, j)\}$

**if** $E \neq alph(w)$ **then return** *true* **else return** *false*
```

In our Algorithm VERSION2 we use rules weaker than Holub's, hence they satisfy the same property as in Fact 4:

Fact 5 *Extending a correct triple using any of the rules (a), (b'), (c'), (d'), (e') leads to a correct triple. In particular, if any sequence of actions corresponding to these rules leads to $E = alph(w)$ then w is morphically primitive.*

Lemma 6. *Assume the execution of Algorithm* VERSION*2 has finished with the triple (E, L, R). Then for each $1 \le i < j \le n$ such that $w[i] \in E$ and $j = succ_E(i)$ we have*

$$i + \gamma_{right}(w[i]) = j - 1 - \gamma_{left}(w[j]). \tag{1}$$

Proof. First note that, under the conditions of the lemma, both $\gamma_{right}(w[i])$ and $\gamma_{left}(w[j])$ are finite. Indeed, the former is finite since $i \in R$ and the latter is finite since the left end of the range of j, that is, $j - 1 - left(j)$, belongs to R.

For a proof by contradiction assume that (1) does not hold for some i, j. Assume first that $i + \gamma_{right}(w[i]) < j - 1 - \gamma_{left}(w[j])$. We have two cases here. Let $u = i + right(i)$ and $v = j - 1 - \gamma_{left}(w[j])$. If $u < v$ then one could perform the operation (e) on $u \in L$ and $v \in R$. However, by taking $u' = pred_L(v)$ and $v' = succ_R(u')$ we obtain a pair of positions $u', v' \in [u, v]$ that would enable us to perform the (e') operation, hence the algorithm has not finished yet. In the opposite case, if $v \le u$, the inter-position $v \in R$ would contradict the definition of $\gamma_{right}(w[i])$.

Now assume that $i + \gamma_{right}(w[i]) > j - 1 - \gamma_{left}(w[j])$. Then the inter-position $i + \gamma_{right}(w[i]) \in R$ contradicts the definition of $\gamma_{left}(w[j])$. □

Theorem 7. *Algorithm* VERSION*2 correctly decides whether w is morphically imprimitive. If so, the corresponding morphic factorization of w is described by the following factors (see Fig. 3):*

$$\{w[i - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])] \,:\, w[i] \in E\}. \tag{2}$$

Proof. Assume that the triple (E, L, R) returned by our algorithm satisfies $|E| < |alph(w)|$. Then, by Lemma 6, the factorization (2) is a nontrivial morphic factorization of w that induces a morphism h such that $h(w) = w$.

Now assume that $|E| = |alph(w)|$ in the final triple of the algorithm. Then, by Fact 5, the word w is morphically primitive. □

4 Efficient version of the algorithm

Algorithm VERSION2 is of a nondeterministic nature, inserting new elements to L, R and E is performed in any order until the configuration stabilizes. Now we proceed to efficient deterministic implementation, which is *neighbour-driven*: the actions are performed locally between neighbours, new elements inserted to R affect their *E-neighbours* which potentially generate new elements of R in *neighbourhoods* (ranges) of occurrences of these E-neighbours. For a position i its left/right E-neighbour is at position $pred_E(i)/succ_E(i)$.

We say that an interval $[i, j]$ is *loose* if the rule (e') from Algorithm Version2 can be applied to (i, j). In other words, an interval is loose if it can generate a new expanding letter $\alpha(i, j)$.

Description of one iteration of the algorithm. In one main iteration Algorithm Version3 adds a single new expanding letter e (the letter is generated using the function *ProcessInterval*). We maintain the set of *loose* intervals, which are potential generators of such letters, in *IntervalsSet* data structure. From now only new inter-positions are inserted to R and L until the configuration stabilizes. This phase corresponds to multiple application of the (most expensive) part (d') in Algorithm Version2. A new expanding letter e, by the rules (b') and (c'), generates a number of new elements of R and L. New elements of R by a *chain-reaction* cause further insertions to R that are related to other letters e', then they affect others and so on. This chain-reaction is performed using the queue *LettersQueue*. Each insertion of a new element to R can affect all occurrences of its both E-neighbours (due to the rule (d')); we insert these E-neighbours into *LettersQueue* to be processed later. We successively dequeue a letter e from *LettersQueue* and compute the set of new elements of R that occur due to the rule (d') applied for any $w[i] = e$. Each new insertion into R is processed by the function *Propagate*(k). It is responsible for insertion into *LettersQueue* of the E-neighbours of k and updating their γ-values.

We proceed now to a more detailed description of the whole algorithm. The algorithm starts with $L = R = \{0, n\}$, $E = \emptyset$ and when it terminates none of the operations (b'), (c'), (d'), (e') can be performed. For each $e \in E$ the values $\gamma_{left}(e)$ and $\gamma_{right}(e)$ are explicitly stored, whereas the values $\gamma_{left}(i)$, $\gamma_{right}(i)$, $left(i)$ and $right(i)$ for $i \in Occ(E)$ are computed on demand using predecessor/successor queries.

The following function *ProcessInterval*(i, j) performs the action (e'), and then the actions (b'), (c') from Algorithm Version2. The function also accounts for the situation when the new expanding letter causes the right range of its E-neighbour to decrease. It returns the set of newly inserted elements of R.

```
Function ProcessInterval($i, j$)
$e := \alpha(i, j)$; $E := E \cup \{e\}$; $NewR := \emptyset$
foreach $p \in Occ(e)$ do
    $NewR := NewR \cup \{p,\ p - left(p) - 1\}$
    $L := L \cup \{p + right(p)\}$
$NewR := NewR \setminus R$; $R := R \cup NewR$
compute $\gamma_{right}(e)$, $\gamma_{left}(e)$; add $e$ to $LettersQueue$
foreach $p \in Occ(e)$ do
    $e' := w[pred_E(p)]$; $\{e'$ is $E$-neighbour of position $p\}$
    if $\gamma_{right}(e') > right(pred_E(p))$ then
        $\gamma_{right}(e') := \max\{pred_R(p' + right(p') + 1) - p' \ :\ p' \in Occ(e')\}$
return $NewR$
```

The function *Propagate* is called each time a new element is inserted to R. It is responsible for inserting into *LettersQueue* and updating the γ-values of expanding letters.

```
Function Propagate($i$)
  {we assume that $i \in R$}
  $e_1 := w[pred_E(i+1)];\ e_2 := w[succ_E(i)]$
  {$e_1, e_1$ are $E$-neighbours of inter-position $i$}
  add $e_1, e_2$ to $LettersQueue$
  update $\gamma_{right}(e_1)$ and $\gamma_{left}(e_2)$ (if necessary)
```

For each letter $e \in E$, we store the set of its occurrences for which the rule (d') would insert a new inter-position to R:

$$ActiveSet(e) = \{i \in Occ(e) : \gamma_{right}(i) < \gamma_{right}(e) \vee \gamma_{left}(i) > \gamma_{left}(e)\}.$$

The implementation of operations required to store these sets is provided in the next section. We additionally need a queue *LettersQueue* that stores elements of E. If any expanding letter e satisfies $ActiveSet(e) \neq \emptyset$ then it is guaranteed to be present in the queue.

```
Algorithm VERSION3
  $L := R := \{0, n\};\ E := \emptyset$
  $IntervalsSet := \{[0, n]\}$
  while $IntervalsSet$ not empty do {Main Iteration}
    let $[i, j]$ be any element of $IntervalsSet$
    $NewR :=$ ProcessInterval($i, j$)
    {New expanding letter $\alpha(i, j)$ has been added to $E$}
    foreach $k \in NewR$ do Propagate($k$)
    while $LettersQueue$ not empty do
      $e := dequeue(LettersQueue)$
      foreach $i \in ActiveSet(e)$ do
        $NewR := \{i + \gamma_{right}(e), i - 1 - \gamma_{left}(e)\} \setminus R$
        $R := R \cup NewR$
        foreach $k \in NewR$ do $Propagate(k)$
  if $E \neq alph(w)$ then return $true$ else return $false$
```

Theorem 8. [Main Result]
The problem of morphic imprimitivity can be solved in linear time.

Proof. Correctness of Algorithm VERSION3 follows from correctness of Algorithm VERSION2. It suffices to note that at the end of execution none of the rules (a), (b'), (c'), (d'), (e') can be applied. Clearly, we apply all the rules (a), (b'), (c') as soon as possible. As for the rules (d') and (e'), if any of the rules can be applied, *LettersQueue* and *IntervalsSet* is non-empty, respectively.

Let us investigate the time complexity of Algorithm Version3. In the next section we show that, after $O(n)$ preprocessing, predecessor/successor queries for L, R and E can be answered in $O(1)$ time, *ActiveSet*-queries can be answered in time proportional to the number of returned positions (plus one) and *IntervalsSet* can be maintained to answer any-element-removal queries in $O(1)$ time. Moreover, we show there that the ranges l_a, r_a can be precomputed in $O(n)$ time and the $\alpha(i,j)$ queries can be answered in $O(1)$ time.

The main point, with respect to the time complexity, is that the work is amortized by the number of occurrences of a newly inserted letter (this concerns ProcessInterval calls and the steps of the main while-loop) and by the number of newly inserted elements of R (this concerns Propagate calls and all the remaining loops of the main algorithm). This yields $O(n)$ time. □

5 Auxiliary data structures

In this section we show how to implement the data structures which support computing letter ranges and $\alpha(i,j)$, predecessor/successor queries and efficiently maintain the sets *IntervalsSet* and *ActiveSet*(e) for each $e \in E$ in the main algorithm.

5.1 Applications of RMQ and LCP

We use two well-known data structures with $O(n)$ preprocessing time and $O(1)$ query time.

Range Minimum Queries (RMQ). We are given an array $A[1..n]$ of integers. This array is preprocessed to answer the following queries: for an interval $[i,j]$ (where $1 \le i \le j \le n$), find any $k_0 \in [i,j]$ such that $A[k_0] = \min\{A[k] : k \in [i,j]\}$. The best known RMQ data structures have $O(n)$ preprocessing time and $O(1)$ query time [4].

Longest Common Prefix Queries (LCP). Let w be a word of length n. For a pair of indices i,j, $1 \le i,j \le n$, we introduce $lcp(i,j)$ as the length of the longest common prefix between $w[i..n]$ and $w[j..n]$. The *lcp* queries can be performed in $O(1)$ time after $O(n)$ time preprocessing [1, 9].

The RMQ data structure allows efficient computing of $\alpha(i,j)$.

Lemma 9. *$\alpha(i,j)$ can be computed in $O(1)$ time after $O(n)$ preprocessing.*

The following lemma follows easily by multiple application of *lcp*-queries to compute r_a, l_a for all $a \in alph(w)$.

Lemma 10. *The values r_a, l_a for all $a \in alph(w)$ can be computed in $O(n)$ time.*

5.2 Answering incremental predecessor/successor queries

As a tool we use the following known problem.

Marked Ancestor Problem: Let T be a rooted tree with n nodes, each of which can be in two states: marked or unmarked. We are to process a sequence of m operations of the following two types:

mark(v): mark node v;

firstmarked(v): return the first marked node on the path from v to the root.

This classical problem can be solved on-line in $O(n + m)$ time, see [3]. This implies an efficient algorithm for answering incremental predecessor/successor queries.

Lemma 11. *A sequence of $O(n)$ predecessor/successor queries for the sets L, R and $Occ(E)$ can be handled in $O(n)$ total time.*

Proof. Note that each of the sets L, R, $Occ(E)$ is a subset of $\{0, \ldots, n\}$ (initially empty) and the only operation performed on these sets is insertion. Hence, a sequence of m predecessor queries on these sets can be performed on-line in $O(n+m)$ time in total with the data structure for the Marked Ancestor Problem. We use a tree T that is a single path of $n + 1$ nodes, insertion to the sets corresponds to the operation *mark* and *firstmarked* returns the predecessor of a node. Successor queries are handled analogously. $\square$

5.3 Computation of ActiveSets

All the operations performed on the respective ActiveSets are provided by the following auxiliary data structure for dynamic storage of maxima.

Decremental-Maxima: We store an array $t[1..m]$ of integers, initially $t[i] = -\infty$ for all i. We support the following types of operations:

increasevalue(i, v): set $t[i] := \max(v, t[i])$;

max$()$: return $\max\{t[i] : i = 1, \ldots, m\}$;

notmaximal$()$: return any i such that $t[i] \neq$ *max*$()$ or **nil** if no such i exists;

reset$()$: set $t[i] = -\infty$ for all $i = 1, \ldots, m$.

Lemma 12. *The Decremental-Maxima data structure can be initialized in $O(m)$ time so that reset$()$ operation is performed in $O(m)$ time and every other operation is performed in $O(1)$ time.*

Proof. We store the array t, the current maximum M and two lists L and L' of elements from $\{1, \ldots, m\}$. Each element belongs to exactly one of the lists and as $p[i]$ we store the pointer to its current location in its list. The list L contains all i for which $t[i] = M$ and L' contains the remaining elements of the domain.

The *max*$()$ operation is performed by returning M. In *increasevalue*(i, v) we update $t[i]$ if $v > t[i]$. If $v = M$ then i is moved from L' to L and if $v > M$ then v becomes M, the list L is appended to L' and L becomes the single element i. Finally, *notmaximal*$()$ returns any element of the list L' if it is non-empty. $\square$

Lemma 13. *All the operations on all ActiveSets can be implemented in $O(n)$ total time.*

Proof. For a given $e \in E$, all the elements $i \in \mathit{ActiveSet}(e)$ can be divided into those for which $\gamma_{\mathit{right}}(i) > \gamma_{\mathit{right}}(e)$ and those for which $\gamma_{\mathit{left}}(i) < \gamma_{\mathit{left}}(e)$. We handle both cases separately. Now we show how to solve the former case with a single instance of the Decremental-Maxima data structure with $m = |\mathit{Occ}(e)|$.

When a new letter e is inserted to E (in ProcessInterval(i, j)), we build the data structure corresponding to e: we apply *reset* and *increasevalue* for all $p \in \mathit{Occ}(e)$ using pred_R queries for the right endpoints of the ranges. This takes $O(|\mathit{Occ}(e)|)$ time for each new letter, $O(n)$ time in total.

Updates of the data structure take place when $\gamma_{\mathit{right}}(p)$ changes for any $p \in \mathit{Occ}(e)$. Whenever a new element i is inserted to R, *increasevalue* may be called only for $\mathit{pred}_E(i)$ (if i is in its range). In total we have $O(n)$ such calls.

Throughout the algorithm γ_{right} may occasionally decrease. This takes place in the last for-all-loop of ProcessInterval(i, j), due to insertion of the new letter e, $\gamma_{\mathit{right}}(e')$ may decrease. In this case we recompute the data structure for e' from scratch in $O(|\mathit{Occ}(e')|)$ time. Let $e'_1, \ldots, e'_l$ be all the letters for which γ_{right} decreased because of the new letter e. Note that each occurrence of each e'_i is a pred_E element of the corresponding occurrence of e. Hence

$$\sum_{i=1}^{l} |\mathit{Occ}(e'_i)| \;\le\; |\mathit{Occ}(e)|.$$

Consequently, this yields a cost of $O(|\mathit{Occ}(e)|)$ per a new letter which gives $O(n)$ time in total.

The only remaining operation on ActiveSets is the "**foreach** $i \in \mathit{ActiveSet}(e)$" loop in the main algorithm. The total number of steps of this loop is $O(n)$ — since each step inserts a new element to R — and selecting any $i \in \mathit{ActiveSet}(e)$ is performed in $O(1)$ time using the *notmaximal* operation.

The case of γ_{left} is similar (we take minimum instead of maximum, also recomputation from scratch is never needed). In conclusion, all the necessary updates and queries on ActiveSets take $O(n)$ total time. □

5.4 Implementation of IntervalsSet

IntervalsSet is needed to generate new expanding letters. It is maintained as a linked list of pairs of integers.

Lemma 14. *All operations on IntervalsSet can be implemented in $O(n)$ total time.*

Proof. We implement the *IntervalsSet* as a linked list of loose intervals. We also store an array that, for each element $i \in \{0, \ldots, n\}$, points to the loose interval in the list that has i as its endpoint (if there is any such loose interval).

At the beginning of the algorithm the list contains a single element $[0, n]$. Each insert operation on sets L and R causes at most one insertion to and at most one deletion from the *IntervalsSet*, the new loose interval is computed using a single predecessor/successor query. Similarly, each insertion of $e \in E$ causes at most one deletion from the *IntervalsSet* per each $p \in Occ(e)$, that is, a deletion of the formerly loose interval.

Consequently, we maintain up-to-date contents of the *IntervalsSet* after each insertion to sets L, R and $Occ(E)$ with $O(1)$-time overhead. $\square$

6 Conclusions

We presented a linear time algorithm for deciding if a word is morphically imprimitive. We started from the original quadratic algorithm by Holub (Algorithm Version1), transformed it by reducing the set of rules used by the algorithm (Algorithm Version2) and finally provided several efficient data structures that enabled linear-time implementation of the previous version if a specific order of performing the rules is applied (Algorithm Version3). Algorithm Version3 tests if a word is morphically imprimitive. It can also provide a morphic factorization (due to Theorem 7).

References

1. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
2. Ehrenfeucht, A., Rozenberg, G.: Finding a homomorphism between two words is NP-complete. Inf. Process. Lett. 9(2), 86–88 (1979)
3. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC) pp. 246–251 (1983)
4. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
5. Head, T.: Fixed languages and the adult languages of OL schemes. International Journal of Computer Mathematics 10(2), 103–107 (1981)
6. Head, T., Lando, B.: Fixed and stationary ω-words and ω-languages. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L. pp. 147–156. Springer-Verlag (1986)
7. Holub, S.: Algorithm for fixed points of morphisms — visualization. http://www.karlin.mff.cuni.cz/~holub/soubory/Vizual/stranka2.html
8. Holub, S.: Polynomial-time algorithm for fixed points of nontrivial morphisms. Discrete Mathematics 309(16), 5069–5076 (2009)
9. Ilie, L., Tinta, L.: Practical algorithms for the longest common extension problem. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE. Lecture Notes in Computer Science, vol. 5721, pp. 302–309. Springer (2009)
10. Matocha, V., Holub, S.: Complexity of testing morphic primitivity. CoRR abs/1207.5690v1 (2012)
11. Reidenbach, D., Schneider, J.C.: Morphically primitive words. Theor. Comput. Sci. 410(21-23), 2148–2161 (May 2009), http://dx.doi.org/10.1016/j.tcs.2009.01.020